Tutorial

Table of contents

1. Overview

The aDORe Archive provides a robust plug-in architecture allowing developers to pre-process content into XMLtapes and ARCfiles prior to indexing and registration. This tutorial exists to help newcomers quickly understand the abilities of the aDORe Archive API. The tutorial should provide basic working knowledge of the API and application configurations. By the end of this process, we'll have a ProfileProcessor implementation which can read-in METS and DIDL documents, identify resource URLs, download resources, write content to an ARC file, and update corresponding references in the active document. Once we've completed the implementation, we'll create a new processing profile configuration set in archive.properties, then import our METS and DIDL content. To minimize complexity, this tutorial will focus on the METS DocProcessor implementation only. Please refer to the code and the DID-API site for additional details regarding DIDL implementations.

2. Defining Our Use Case

Before we can go any further, we need to define the problem we intend to solve. For this tutorial, lets use the following scenario as our use case:
We have significant collection of METS objects. Each object is represented in a single METS document. The METS documents each contain a MODS metadata section, as well as a FLocat element containing a URL. For an example of our METS document structure, see mets1.xml. We'd like to archive the collection and implement a standards-based digital object repository upon which we can build new services. To ingest our existing content, we'll need to read in each METS document, identify the appropriate resource URIs, then harvest and write resources to one or more ARCfiles. We'll also need to update the METS document with the new resource URL and write the documents to one or more XMLTapes.

Figure 1

3. Creating a Content Pre-Processor

Note: Source code for this tutorial is available within the adore-examples distribution.

To implement the content processor we've structured above, we need to create the following classes:

- TutorialProcessor - Implements the aDORe Archive Profile Processor. The class defines the pre-ingestion process for source xml content.
- TutorialArcWriter - Wraps up the resource processing operations. Resource URLs retrieved from a doc are resolved to a byte array then written to the specified ARCFileWriter instance. Once a resource has successfully been written, the resource's

ARCFile Resolver URL is returned.
- DocProcessor - A simple interface upon which pre-processing implementations can be created. This tutorial provides examples of a METS and DIDL implementations.
- MetsProcessor - Implements the DocProcessor interface developed for the tutorial. The implementation uses the METS Toolkit, developed by Harvard, to process METS documents for ingestion into the aDORe Archive.

4. Download Tutorial Source

Click here to download the project described in this tutorial.

5. TutorialProcessor

First we'll create our ProfileProcessor implementation. The adoreArchive will initialize our implementation using the following methods:

- setArchiveConfig() - passes all the properties defined in the archive.properties file to your implementation. The common aDORe properties have standard API calls, otherwise use getProperties to work with a standard Java Propeties object.
- setArchiveProfile() - passes the constructed ArchiveProfile object to your implementation. This is a convenience object containing the various profile processing properties defined in archive.properties (i.e. mets.ConverterClass=gov.lanl.adore.mets.MetsProcessor)
- setSourceArray() - passes a List of File Objects to be processed, along with their type. We'll only be working with TYPE_XML in this tutorial. We'll be creating the ARCfiles within our implementation.

Once initialized, the runIt() method is called to start processing. The adoreArchive application defers all pre-processing to our implementation and expects us to return a list of XMLtape and ARCfile File objects.

Our implemenation will do the following:

- Implement the ProfileProcessor interface
- Create a new interface, DocProcessor, for processing differing formats (i.e. METS, DIDL)
- Prime our XMLTape & ARCFile Writers
- Pre-process Document using DocProcessor implementation
- Write documents to XMLTape
- Add generated XMLTapes & ARCFile to list of items to be registered

The source for TutorialProcessor will be similar to

```
public class TutorialProcessor implements ProfileProcessor {

    static Logger log =
Logger.getLogger(TutorialProcessor.class.getName());

    private ArrayList<File> arcFileList = new ArrayList<File>();
    private ArrayList<File> xmlTapeList = new ArrayList<File>();
    private ArrayList inputArray;
    private ArchiveConfig archiveConfig;
    private ArchiveProfile archiveProfile;
    private ArchiveIO archiveIO;
    private SingleTapeWriter writer;
    private ARCFileWriter arcw;
    private String arcName;
    private String tapeName;

    /**
     * Gets list of ARCfile File Objects for indexing & registration
     *
     * @return ArrayList of File objects for post-processing
     */
    public ArrayList<File> getArcFileList() {
        return arcFileList;
    }

    /**
     * Gets list of XMLtape File Objects for indexing & registration
     *
     * @return ArrayList of File objects for post-processing
     */
    public ArrayList<File> getXMLTapeList() {
        return xmlTapeList;
    }

    /**
     * Sets ArchiveConfig Properties
     *
     * @param config
     *            ArchiveConfig for adore archive
     */
    public void setArchiveConfig(ArchiveConfig config) {
        this.archiveConfig = config;
    }

    /**
     * Sets ArchiveProfile Pre-processing Properties
     *
     * @param profile
     *            ArchiveProfile for collection pre-processing
     */
    public void setArchiveProfile(ArchiveProfile profile) {
        this.archiveProfile = profile;
    }
```

```
    /**
     * Sets list of files to be processed
     *
     * @param type
     *            Content Type Alias
     * @param sourceFiles
     *            List of files to be preprocessed
     */
    public void setSourceArray(int type, ArrayList sourceFiles) {
        if (type == ArchiveConstants.TYPE_XML) {
            this.inputArray = sourceFiles;
        }
    }

    /**
     * Main processing method called by application. This implementation
     * initializes the relevant properties objects, initializes the XMLTape
and
     * ARCFileWriters, then iterates through the source input. Once the
XMLTapes
     * and ARCFiles have been generated, the files are added to the list of
     * items to be registered by the application.
     */
    public void runIt() throws Exception {
        // Initialize Archive IO, ArcProperties, TapeProperties
        this.archiveIO = new ArchiveIO(archiveConfig);
        TapeProperties.load(archiveConfig.getProperties());
        ARCProperties.load(archiveConfig.getProperties());
        TransProperties.load(archiveConfig.getProperties());
        log.debug("Initialized Properties");

        // Prime Tape & ARC File Names
        tapeName = UUIDFactory.generateUUID().toString().substring(9);
        arcName = UUIDFactory.generateUUID().toString().substring(9);

        // Open Tape Writer
        writer = archiveIO.openSingleTapeWriter(tapeName);
        writer.setXmlTapeID(TapeProperties.getLocalXmlTapePrefix() +
tapeName);
        writer.setArcFileID(ARCProperties.getLocalArcFilePrefix() +
tapeName);
        writer.writeDefaultAdmin();
        log.debug("Initialized XML Tape Writer");

        // Open ARC Writer
        arcw = archiveIO.openARCFileWriter(arcName);
        log.debug("Initialized ARC File Writer");

        // Begin Processing Input
        for (Iterator i = inputArray.iterator(); i.hasNext();) {
            String sourceFile = ((File) i.next()).getAbsolutePath();
            log.debug("Processing " + sourceFile);
            processDocument(sourceFile);
```

```
        }

        // Close tape and arc writers
        writer.close();
        arcw.close();

        // Populate XMLtape & ARCfile lists
        String tapeDir = archiveConfig.getTapeStoredDirectory();
        String arcDir = archiveConfig.getARCStoredDirectory();
        xmlTapeList.add(new File(tapeDir, tapeName + ".xml"));
        arcFileList.add(new File(arcDir, arcName + ".arc"));
    }

    /**
     * Processes each document extracting resource references and updating
the
     * associated doc reference. The DocProcessor interface was introduced
to
     * accommodate multiple document formats (i.e. METS, DIDL, etc).
     *
     * @param sourceFile
     *            File containing a single XML document
     * @throws Exception
     */
    private void processDocument(String sourceFile) throws Exception {
        String plugin = archiveProfile.getCollectionConverterPlugin();
        log.debug("Initializing Converter Plug-in: " + plugin);
        DocProcessor dp = (DocProcessor)
Class.forName(plugin).newInstance();
        dp.setARCOutput(arcw, archiveConfig.getARCResolverURL() +
"?res_id="
                + archiveConfig.getARCFilePrefix() + arcName);

        try {
            dp.read(new FileInputStream(new File(sourceFile)));

            // Write Resources to ARC
            dp.process();

            // Get XML String of Document
            String metadata = dp.write();

            String date = dp.getDate();
            String id = dp.getId();
            if (metadata != null) {
                writer.writeRecord(new TapeRecord(id, date, metadata));
            } else {
                throw new Exception(
                        "Error processing record: No metadata found in
document");
            }
        } catch (FileNotFoundException e) {
            log.error(e.getClass().getName() + " occurred for " +
sourceFile);
```

```
            throw new Exception("Error processing " + sourceFile + ": "
                    + e.getMessage());
        } catch (Exception e) {
            log.error(e.getMessage());
            throw new Exception("Unable to process " + sourceFile + ": "
                    + e.getMessage());
        }
    }
}
```

You'll notice that in processDocument() we define a plug-in for the DocProcessor interface. Next we'll create this interface and our METS implementation.

6. DocProcessor

The DocProcessor will provide a simple interface upon which pre-processing implementations can be created. Since we'd like to support the processing of many different metadata formats (i.e. METS, DIDL, etc.), each implementation will need to support the following:

- Implement this interface
- Read in the source document from an InputStream
- Implement an open-ended processing method
- Serialize processed document to a valid XML String
- Implement methods to obtain identifier and datestamp for object
- Pass ARCFileWriter and ARCFileResolver BaseUrl to implementing class

The source for DocProcessor will be similar to:

```
public interface DocProcessor {

    /**
     * Reads in an InputStream for processing
     *
     * @param source
     *            InputStream contain document to process
     * @throws TutorialException
     */
    public abstract void read(InputStream source) throws TutorialException;

    /**
     * Performs any pre-processing routines defined by the implemenation.
     *
     * @throws TutorialException
     */
    public abstract void process() throws TutorialException;

    /**
```

```
     * Serializes processed document to XML String format
     *
     * @return Escaped XML String instance of document
     * @throws TutorialException
     */
    public abstract String write() throws TutorialException;

    /**
     * Gets the last modified date of the TapeRecord. For example:
     * 2006-03-07T12:00:00Z
     *
     * @return UTC Date as String
     */
    public abstract String getDate();

    /**
     * Gets local repository id for document (i.e. OAI-Identifier). For
example:
     * info:lanl-repo/i/dd7b17ea-bddf-11d9-9de5-c11b6cd8559
     *
     * @return URI as String
     */
    public abstract String getId();

    /**
     * Sets the ARCFileWriter and ARC Resolver Base Url. These are used by
     * pre-processing implementation if you wish to harvest referenced
resources
     * to store locally in ARCFiles.
     *
     * @param arcw
     *            Initialized ARCFileWriter instance
     * @param arcUrl
     *            BaseUrl to ARC File Resolver instance
     */
    public abstract void setARCOutput(ARCFileWriter arcw, String arcUrl);

}
```

Now that we've defined our DocProcessor interface, we can create our METS implementation.

7. MetsProcessor

The MetsProcessor will utilize the METS Toolkit library, developed by Harvard, to process each document.

Our implementation will do the following:

- Implement the DocProcessor interface
- Read InputStream to create Mets object

- Get Object ID from Mets object
- Create wrapper class, TutorialArcWriter, to handle resource dereferencing and arc file serialization
- Iterate through content objects to obtain FLocat elements
- Pass obtained URLs to TutorialArcWriter
- Update corresponding FLocat element in Mets object
- Create UTC Datestamp to reflect last modified date
- Serialize Mets object as valid XML String

The source for MetsProcessor will be similar to:

```
public class MetsProcessor implements DocProcessor {

    static Logger log = Logger.getLogger(MetsProcessor.class.getName());

    private Mets mets;
    private TutorialArcWriter arcWriter;
    private String id;
    private String date;

    /**
     * Reads in an InputStream for processing. The MetsReader is used to
parse
     * and validate document.
     *
     * @param source
     *             InputStream contain document to process
     * @throws TutorialException
     */
    public void read(InputStream source) throws TutorialException {
        mets = new Mets();
        try {
            MetsReader r = new MetsReader(source);
            mets.read(r);
            mets.validate(new MetsValidator());

            // Gets value used for tape record id
            id = mets.getOBJID();
            log.debug("Processing " + id);
        } catch (MetsException e) {
            throw new TutorialException("Error processing METS document: "
                    + e.getMessage());
        } catch (Exception e) {
            throw new TutorialException("Error processing source "
                    + e.getMessage());
        }
    }

    /**
     * Iterates through content objects to obtain FLocat elements. URLs
obtained
```

```
     * from the FLocat element are resolved and written to the initialized
     * ARCFileWriter instance. Once written, the ARC File Resolver URL to
the
     * resource is returned. The FLocat ref is updated in the document.
     */
    public void process() throws TutorialException {
        for (Iterator i = mets.getContent().iterator(); i.hasNext();) {
            MetsElement me = (MetsElement) i.next();
            if (me.getQName().equals("fileSec")) {
                FileSec fs = (FileSec) me;
                for (Iterator j = fs.getContent().iterator(); j.hasNext();)
{
                    me = (MetsElement) j.next();
                    if (me.getQName().equals("fileGrp")) {
                        FileGrp fg = (FileGrp) me;
                        for (Iterator k = fg.getContent().iterator(); k
                                .hasNext();) {
                            me = (MetsElement) k.next();
                            if (me.getQName().equals("file")) {
                                File file = (File) me;
                                try {
                                    for (Iterator l = file.getContent()
                                            .iterator(); l.hasNext();) {
                                        me = (MetsElement) l.next();
                                        if (me.getQName().equals("FLocat"))
{
                                            FLocat loc = (FLocat) me;
                                            log.debug(file.getMIMETYPE()
                                                    + " | "
                                                    + loc.getXlinkHref());
                                            String arcUrl =
arcWriter.write(loc
                                                    .getXlinkHref(), file
                                                    .getMIMETYPE());
                                            loc.setXlinkHref(arcUrl);
                                        }
                                    }
                                } catch (ARCException e) {
                                    throw new TutorialException(
                                            "Error processing " +
file.getID()
                                                    + ": " +
e.getMessage());
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    /**
     * Serializes processed document to XML String format
```

```
     *
     * @return Escaped XML String instance of document
     * @throws TutorialException
     */
    public String write() throws TutorialException {
        date = DateUtil.date2UTC(new Date());
        log.debug("Date: " + date);
        ByteArrayOutputStream out;
        try {
            mets.validate(new MetsValidator());
            out = new ByteArrayOutputStream();
            mets.write(new MetsWriter(out));
        } catch (MetsException e) {
            throw new TutorialException(e.getMessage());
        }
        return format(out.toString());
    }

    /**
     * Gets the last modified date of the TapeRecord. For example:
     * 2006-03-07T12:00:00Z
     *
     * @return UTC Date as String
     */
    public String getDate() {
        return date;
    }

    /**
     * Gets local repository id for document (i.e. OAI-Identifier). For
example:
     * info:lanl-repo/i/dd7b17ea-bddf-11d9-9de5-c11b6cd8559
     *
     * @return URI as String
     */
    public String getId() {
        return id;
    }

    /**
     * Should the provided xml content contain an xml declaration, will
strip
     * off the declaration and return doc
     */
    private static String format(String xml) {
        if (xml.startsWith("<?")) {
            int end = xml.indexOf("?>");
            xml = xml.substring(end + 2);
        }
        return xml;
    }

    /**
     * Sets the ARCFileWriter and ARC Resolver Base Url. These are used by
```

```
     * pre-processing implementation if you wish to harvest referenced
resources
     * to store locally in ARCFiles.
     *
     * @param arcw
     *            Initialized ARCFileWriter instance
     * @param arcUrl
     *            BaseUrl to ARC File Resolver instance
     */
    public void setARCOutput(ARCFileWriter arcw, String arcUrl) {
        arcWriter = new TutorialArcWriter(arcw, arcUrl);
    }

}
```

You'll notice we created a new class, TutorialArcWriter, to help us manage resource dereferencing and ARCfile serialization. Next we'll create this ARCfile helper class.

8. TutorialArcWriter

Our helper class with do the following:

- Download Resource from a URL
- Write byte[] of resource to ARCfile
- Generate ARC File Resolver URL for written resource

The source for TutorialArcWriter will be similar to:

```
public class TutorialArcWriter {

    private String arcurl;

    private ARCFileWriter arcwriter;

    /**
     * Initializes a new TutorialArcWriter instance for the defined
     * ARCFileWriter and ARCFileResolver BaseUrl
     *
     * @param arcwriter
     *            Initialized ARCFileWriter instance
     * @param arcurl
     *            BaseUrl of ARC File Resolver web-app
     */
    public TutorialArcWriter(ARCFileWriter arcwriter, String arcurl) {
        this.arcwriter = arcwriter;
        this.arcurl = arcurl;
    }

    /**
     * Writes resource located at defined url to initialized ARCFileWriter
```

```
     * instance.
     *
     * @param url
     *            URL of resource to be resolved and written to an arc file
     * @param arcmimetype
     *            Mimetype of the resource to be resolved
     * @return ARC File Resolver URL to written resource
     * @throws ARCException
     */
    public String write(String url, String arcmimetype) throws ARCException
{
        String arcid = TransProperties.getLocalDataStreamPrefix()
                + UUIDFactory.generateUUID().toString().substring(9);
        byte[] deref;
        try {
            deref = resolveRef(url);
        } catch (MalformedURLException e) {
            throw new ARCException(e.getMessage());
        } catch (IOException e) {
            throw new ARCException(e.getMessage());
        }
        arcwriter.write(arcid, "0.0.0.0", arcmimetype, deref);
        return composeRef(arcid);
    }

    /**
     * Composes an ARC File Resolver URL for the provided identifier
     *
     * @param id
     *            Identifier of the resource
     * @return ARC File Resolver URL to written resource
     */
    public String composeRef(String id) {
        String ref = arcurl + "&rfr_id="
                + TransProperties.getLocalOpenUrlReferrerID()
                + "&url_ver=Z39.88-2004&rft_id=" + id;
        return ref;
    }

    /**
     * Gets specified resource from remote URL as byte array
     *
     * @param url
     *            Location of the resource to resolve
     * @return Btye Array instance of resource
     * @throws MalformedURLException
     * @throws IOException
     */
    public static byte[] resolveRef(String url) throws
MalformedURLException,
            IOException {
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        URL addressurl = new URL(url);
```

```
        InputStream in = addressurl.openStream();
        BufferedReader bin = new BufferedReader(new InputStreamReader(in));

        int bufferSize = 4096;
        byte[] buffer = new byte[bufferSize];
        int bytesRead;

        while ((bytesRead = in.read(buffer, 0, bufferSize)) != -1) {
            out.write(buffer, 0, bytesRead);
            out.flush();
        }

        return out.toByteArray();
    }
}
```

Now that we've defined our implementation, lets build our project and configure the aDORe Archive application to recognize our new processor.

9. Building AdoreExample

By this point, your Adore Archive installation should be up and running. If you're unsure, follow the steps outlined in the Demonstration page.

1. Let's open the module.conf file for the AdoreExamples project; locate the adoreArchive.home property. Set this property as the file path to the adore-archive directory (i.e. /usr/local/adoreArchive/adore-archive).

2. To build our project, run "ant build" from the AdoreExamples project directory.

3. To deploy our implementation and dependent libraries, run "ant deploy" from the AdoreExamples project directory. Deploy will also copy the contents of the etc directory to a new 'tutorial' directory created in the adore-archive/etc directory.

Now our ProfileProcessor implementation and dependent libraries are available to the aDORe Archive application. Next we need to define our setSpec XPath properties and register our new "mets" processing profile and its associated components.

10. Creating a SetSpec XPath Properties File (optional)

The SetSpec XPath Properties file defines the XPath locations of elements you wish to include as setSpecs. During the indexing process, each XPath definition contained in the properties file will be used to look-up setSpec values. Within the aDORe Archive environment, setSpec prefixes are used to support setSpecs of differing types. The most common types are 'collection' and 'format'. The setSpec prefixes are OPTIONAL values. When setSpec prefixes are used, the prefix and value are delimited by a colon (i.e.

collection:tutorialCollection). Also, it should be noted that all setSpec values are URLEncoded and percent signs (%) are replaced with asterisks (*). This means that a setSpec value of 'info:sid/library.lanl.gov:demo' will be indexed as 'info*3Asid*2Flibrary.lanl.gov*3Ademo'; this ensures OAI setSpec compliance. For our sample METS document, let's say you'd like to use information from the MODS record as setSpec values. You might want to use the mods:publisher element as a collection setSpec value and mods:internetMediaType as a format setSpec value.

A SetSpec XPath Properties file matching our criteria would be similar to:

```
# METS SetSpec Processing Profile
profile.name=mets

# Namespace Definitions
profile.namespace.1=http://www.loc.gov/METS/
profile.namespace.prefix.1=mets
profile.namespace.2=http://www.loc.gov/mods/v3
profile.namespace.prefix.2=mods

# XPath Definitions
profile.xpath.1=//mets:xmlData/mods:originInfo/mods:publisher
profile.xpath.prefix.1=collection
profile.xpath.2=//mets:xmlData/mods:physicalDescription/mods:internetMediaType
profile.xpath.prefix.2=format
```

To simplify the tutorial, the AdoreExamples Ant Deploy task copies a version of this file to the 'etc/tutorial/conf/IndexSetSpecProps' directory of the adore-archive installation. You'll just need to update the path in the processing profile. Next we'll create and add our processing profile to the archive.properties file.

11. Configuring archive.properties

Now that we've defined our ProfileProcessor implementation and created a setSpec XPath Properties file, let's register our METS processor in the archive.properties file. The archive.properties file defines environmental variables used by the aDORe Archive. The adoreArchive installer creates a pre-configured instance of archive.properties in the 'adore-archive/etc' directory (i.e. /usr/local/adoreArchive/adore-archive/etc/archive.properties).

Open archive.properties file and add the following to the bottom:

```
# METS Processing Profile
mets.FullName=info:sid/library.lanl.gov:metsCollection
```

```
mets.ProcessorClass=gov.lanl.adore.demo.TutorialProcessor
mets.ConverterClass=gov.lanl.adore.demo.mets.MetsProcessor
mets.IndexSetSpecProps=/usr/local/adoreArchive/adore-archive/etc/tutorial/conf/IndexSet
```

You might need to update the path to the IndexSetSpecProps file depending upon your installation. A copy of the file fragment described above can be found in the 'etc/conf/ArchiveProps/' directory of the AdoreExamples project.

What did we just configure? We have just registered our recently created processing profile in the adore-archive application. Now when we want to ingest content from this collection, all we need to do is specify the processing profile alias (i.e. mets) and which files to process. The application will then know to use gov.lanl.adore.demo.TutorialProcessor as our base plug-in and gov.lanl.adore.demo.mets.MetsProcessor for processing the source documents. Once our content has been transformed and appended to an XMLtape, the IndexSetSpecProps file is used to index setSpec values contained in each TapeRecord.

12. Running aDORe Archive using our new processor

In the bin directory of the adore-archive module is the adoreArchive.sh Shell Script. This script and its batch file compatriot are the gatekeepers for the aDORe Archive ingestion process.

Each of the scripts expect the following input:

- --config [archiveProp] - Path to archive.properties file [REQUIRED]
- --profile [collectionPrefix] - Profile prefix as defined in archive properties file [REQUIRED]
- --xml || --xmltape [xmlContentPath] - Path to dir/file to process [OPTIONAL]
- --arcfile [arcFilePath] - Path to dir/file to process [OPTIONAL]

Since our processor implementation will be creating the ARCfiles, we just need to let the application know 1) the location of our archive.properties file, 2) that our processing profile alias is "mets", and 3) the location of the METS documents we'd like to ingest. For the --xml and --arcfile arguments, we can either specify a single file or a directory containing multiple files. You also have the option to add the --recursive flag to recursively process all files matching the appropriate file filter, ".xml" or ".arc".

From adoreArchive/adore-archive/bin, run the following:
sh ./adoreArchive.sh --config ../etc/archive.properties --profile mets --xml ../etc/tutorial/data/mets
Once the ingestion process has completed, you should see a section listing the newly registered XMLtapes and ARCfiles. To check the contents of your newly created XMLtape, take the UUID part of the XMLtape file name (i.e. the

f796027b-e403-4194-b148-fda0225e20d8 part of f796027b-e403-4194-b148-fda0225e20d8.xml) and add it onto the base of your Adore Archive Accessor instance. For instance, if the Base URL is "http://localhost:8080/adore-archive-accessor/Handler/", add the UUID to the end (i.e. http://localhost:8080/adore-archive-accessor/Handler/f796027b-e403-4194-b148-fda0225e20d8) then add the standard OAI-PMH arguments. For this example, http://localhost:8080/adore-archive-accessor/Handler/f796027b-e403-4194-b148-fda0225e20d8?verb=ListRecords&metadataPrefix=native will return the list of all records contained in that XMLtape. So your looking at the "native" metadataPrefix and thinking "What's that?". Well, currently the adore-archive-accessor is intended to simply pull-out the contents of a TapeRecord regardless of the metadataFormat. The use of "native" makes no assumptions to the metadata contained in the record. You can add support for alternate metadataFormats by adding a new Crosswalks mapping to the moai.properties file located in the WEB-INF directory of the adore-archive-accessor web app. Please note that adding "Crosswalks.mets=gov.lanl.xmltape.oai.TapeCrosswalk" will function identically to native.